ACM SIGACT News Distributed Computing Column 3

Sergio Rajsbaum*

May 15, 2001

Abstract

The Distributed Computing Column covers the theory of systems that are composed of a number of interacting computing elements. These include problems of communication and networking, databases, distributed shared memory, multiprocessor architectures, operating systems, verification, internet, and the web.

After some announcements, this issue includes a contribution by Idit Keidar and myself.

Request for Collaborations: Please send me any suggestions for material I should be including in this column, including news and communications, and suggestions for authors willing to write a guest column or to review an event related to theory of distributed computing. In particular, I would welcome open problems with a short description of context and motivation.

Announcements

- PODC 2001 will be held **August 26–29, 2001** in **Newport, Rhode Island**. Go to `http://www.podc.org/podc2001/` for up to date announcements. The list of accepted papers is available at `http://www.podc.org/podc2001/accepted.html`
- In honor of his sixtieth birthday, PODC 2001 will feature a series of lectures illustrating and celebrating the vast impact of the work of **Leslie Lamport**. The following are the invited speakers in this lecture series: Martin Abadi (InterTrust), Jim Anderson (University of North Carolina at Chapel Hill), Maurice Herlihy (Brown University), Butler Lampson (Microsoft), Nancy Lynch (MIT), Amir Pnueli (The Weizmann Institute), Chris Rowley (Open University,UK and LaTeX3 Project). This lecture series is being organized by Cynthia Dwork of Compaq.
- The PODC Steering Committee is pleased to announce that the chair of the Program Commitee for PODC 2002 will be **Keith Marzullo**. Dr. Marzullo is Associate Professor of Computer Science at the University of California at San Diego. PODC 2002 will be held in the **San Francisco Bay area**. The exact location and dates are still to be determined.
- To celebrate twenty years of PODC, the journal Distributed Computing is planning to publish a special issue, featuring surveys, retrospectives, and personal perspectives. More information is available at `http://www.cs.technion.ac.il/~hagit/podc20/`

*Compaq Cambridge Research Laboratory, One Cambridge Center, Cambridge, MA 02142-1612. Sergio.Rajsbaum@compaq.com, rajsbaum@math.unam.mx. On leave from Instituto de Matemáticas, UNAM.

- The workshop PERSPECTIVES ON ALGORITHMS AND DISTRIBUTED ALGORITHMS is intended to explore new problems and future trends in algorithms. The workshop is organized around the lectures of 11 distinguished speakers and extensive discussion sessions. It will be held at the Centre International de Rencontres Mathematiques in Luminy, France, May 21 to 24, 2001. `http://www.cirm.univ-mrs.fr`

On the Cost of Fault-Tolerant Consensus
When There Are No Faults
(Preliminary Version)

Idit Keidar [1] and Sergio Rajsbaum[2]

Abstract

We consider the consensus problem in an asynchronous model enriched with unreliable failure detectors and in the partial synchrony model. We consider algorithms that solve consensus and tolerate crash failures and/or message omissions. We prove tight lower bounds on the number of communication steps performed by such algorithms in failure-free executions. We present in a unified framework a number of related lower bound results. Thus, we shed light on the relationships among different known lower and upper bounds, and at the same time, illustrate a general technique for obtaining simple and elegant lower bound proofs. We also illustrate matching upper bounds: we algorithms that achieve the lower bound.

1 Introduction

Consensus is a fundamental problem in distributed computing theory and practice alike. A consensus service allows a collection of processes to agree upon a common value. More specifically, each process has an input, and each correct process must *decide* on an output, such that all correct processes decide on the same output, and furthermore, this output is the input of one of the processes. Consensus is an important building block for fault-tolerant distributed systems [22]: to achieve fault-tolerance data is replicated, and consensus can be used to guarantee replica consistency using the *state machine* approach [21, 27].

Consensus is not solvable in pure asynchronous models where even a single process can crash [13]. However, consensus is solvable in models of partial synchrony [11], where the system can be asynchronous for an unbounded but finite period of time, but then eventually becomes synchronous, as long as less than a majority of the processes can crash. Similarly, consensus is solvable in an asynchronous model enriched with certain oracle failure detectors [6], including unreliable failure detectors that can provide arbitrary output for an arbitrary period of time, but eventually provide some useful semantics. We illustrate this in Section 3. In this paper we are interested in the cost of consensus algorithms in these models – partial synchrony and asynchronous with unreliable failure detectors.

[1]MIT Laboratory for Computer Science
[2]Compaq CRL and UNAM

We consider two failure models: crash failures and message omissions, where correct processes can lose messages. Consensus is not solvable in the omission failure model, even in a synchronous system, unless some restrictions are placed on possible message loss patterns. In particular, consensus is not solvable even in a single mobile failure model where the environment (the adversary) can only choose one process in each communication step of the algorithm, and lose some of the messages sent in this step by this process [25]. We prove this impossibility result in Section 4. The impossibility result relies on the adversary's ability to choose *any* process to lose messages from in every step. In order to make consensus solvable, the adversary can be restricted in a number of different ways. For example, if the adversary is restricted so that eventually there is one particular process it cannot "touch", that is, messages from this designated process cannot be lost from some point on, and in addition, messages from less than a majority of the processes can be lost in each communication step, then consensus is solvable [19]. We illustrate this in Section 3.

Now, let us examine the running time of consensus algorithms. Fault-tolerant consensus algorithms, even for the synchronous model, are bound to have executions in which they take a long time to terminate. The worst-case running time of a consensus algorithm tolerating crash failures in a synchronous model grows linearly with the number of failures it can tolerate. Specifically, a consensus algorithm that can tolerate up to t crash failures will have executions involving a sequence of $t + 1$ communication steps [10, 23]. In other words, if the network latency is bounded by δ, any algorithm will have executions where it takes at least $(t + 1)\delta$ time for all processes to decide.

In the models we focus on here – partial synchrony and asynchronous with unreliable failure detectors – the situation is even worse. Since the network can be asynchronous for an unbounded period of time, and the failure detectors can provide arbitrary output for an unbounded period of time, the worst-case running time of any consensus algorithm in these models is unbounded.

In practice, however, consensus algorithms can reach a decision quickly in common cases. In practical networks there are extensive periods during which communication is timely, even if there is no a priori known upper bound on message latency. Thus, many executions are actually synchronous. In such executions, failure detectors based on time-outs can be accurate. Moreover, in practice, failures are relatively uncommon. It is therefore interesting, from a practical perspective, to study and improve the running time of consensus algorithms under such benign circumstances.

For this purpose, *early-stopping* consensus algorithms were designed. In the synchronous crash failure model, early-stopping consensus algorithms can reach a decision in a time that is linear in the number of failures that actually happen in an execution. In an execution with f failures, early-stopping algorithms can have all the processes decide within at most $f+1$ communication steps [10]. In particular, when there are no failures, these algorithms decide within one communication step.

What can we say about early-stopping consensus algorithms for the partial synchrony and unreliable failure detector models? As explained above, even in the absence of failures, such algorithms are bound to have unbounded running times since the network and failure detectors can behave in an arbitrary way for an unbounded prefix of an execution. However, we are interested in executions in which, from the very beginning of the execution, the network is synchronous or the failure detector is accurate. We will call such executions *well-behaved* if they are also failure-free. Since well-behaved executions are common in practice, algorithm performance under such circumstances is significant. Note, however, that an algorithm cannot know a priori that an execution is going to be well-behaved, and thus cannot rely upon it. We are therefore interested in how well can algorithms that do tolerate failures and partial synchrony (or unreliable failure detection) perform in well-behaved executions.

There are several algorithms for fault-tolerant consensus in partial synchrony and asynchronous with unreliable failure detector models that tolerate crash failures and execute only two communication steps in well-behaved executions (e.g., [26, 16, 24]). In other words, in well-behaved executions in which the network delivers every message within δ time, these algorithms take 2δ time. Distributed systems folklore suggests that every fault tolerant algorithm in "practical" asynchronous or partial synchrony models must take at least two communication steps before deciding, even in well-behaved executions. This is in contrast to consensus algorithms in the synchronous crash failure model, where algorithms can have all processes decide in one communication step in the absence of failures [10].

In this paper we formalize this folklore theorem: we show that the bound holds as long as two or more processes can crash. We show that the need for an additional communication step (over what is needed in the synchronous model) stems from the fact that in the asynchronous and partially synchronous models a correct process can be mistaken for a faulty one. This requires consensus algorithms for these models to avoid disagreement with processes that seem faulty. In contrast, consensus in the synchronous model requires only that correct processes agree upon the same value, and allows for disagreement with faulty ones. The uniform consensus problem strengthens consensus to require that every two processes (correct or not) that decide must decide on the same value. Interestingly, uniform consensus requires two communication steps in the absence of failures even in the synchronous model, as long as two or more processes can crash [8]. The two communication step lower bound for consensus in asynchronous and partial synchrony models stems from the fact that any algorithm that solves consensus in these models, also solves uniform consensus [15].

Interestingly, if we consider a message omission model where an unbounded number of messages from a correct process may be lost, then even in the synchronous model, any algorithm for consensus also solves uniform consensus. Thus, the two communication step bound holds for the synchronous message omission model, as also observed by Lamport [20].

In this paper we provide the intuition and formalize the lower and upper bounds. We illustrate a simple technique to derive several lower bounds in the same framework. In Section 2, we present formal definitions of the models and problems. In Section 3, we illustrate the upper bound; we describe an algorithm that works in two communication steps in well-behaved executions. In order to derive the lower bound, we first consider the synchronous model: in Section 4, we discuss lower bounds for the uniform consensus problem in the synchronous model. Then, in Section 5, we derive a lower bound for consensus in models of partial synchrony and in the asynchronous model with unreliable failure detectors. Finally, Section 6 concludes the paper and discusses optimistic alternatives to consensus.

2 Definitions

2.1 Models

The universe consists of n processes, $p_1, \ldots, p_n$. Processes communicate by message-passing. An execution is a sequence of steps taken by distinct processes. A process can send messages to several other processes in one step.

We consider two failure models: crash failures and message omission. We do not consider Byzantine failures.

In the *crash failures* model, processes fail by permanently crashing. A parameter t bounds the number of processes that can crash. A process that crashes in an execution is *faulty* in that execution, a process that does not crash is *correct*. In this model, communication among correct processes is reliable. That is, every message sent from one correct process to another correct process eventually reaches its destination. If a process fails during a given step, any subset of the messages it sent in this step can be lost. Messages it sent in previous communication steps cannot be lost.

In the *omission failure* model, messages sent by correct processes can be lost in any communication step. A process is *faulty* in this model if there is a point in the execution after which all the messages from this process are lost. If message omission failures are possible, consensus is impossible even for a very restricted form of omission failures. The *single mobile failure model* [25] is a synchronous model where in a step the messages of at most one process can be lost.

We consider three timing models: synchronous, partial synchrony, and asynchronous with unreliable failure detectors.

2.1.1 Synchronous

In the *synchronous* model, processes execute in synchronous steps[3]. In each step, every processes can send messages to any number of other processes. Unless a failure occurs, each message reaches its destination in the following step. We denote by δ the time it takes to execute a step. For simplicity, we assume that δ is the network latency, and the local computation time is zero.

2.1.2 Partial synchrony

There are several *partial synchrony* models in the literature. We consider one of the models defined in [11], where there is no a priori bound on message latency. However, there is a time, called *global stabilization time (GST)* after which there is a bound on message latency, but this bound is not known to the processes. For simplicity, we assume that local computation time is zero and that processes can measure time accurately[4].

In addition, we assume that there exist a pre-defined constant δ that is known to the processes and reflects the typical message latency in the system. A *well-behaved* execution in the partial synchrony model is an execution in which there are no failures and every message reaches its destination within at most δ time[5]. Thus, a well-behaved execution is essentially synchronous.

2.1.3 Asynchronous with failure detectors

In the *asynchronous model with unreliable failure detectors* [6], there is no bound on message latency, and every process p is equipped with a failure detector oracle. At every step of the algorithm's execution, p can query its failure detector for some information. We focus here on failure detectors that can provide arbitrary outputs during an unbounded period of time, but eventually have to provide some meaningful semantics. Our lower bound will apply to any failure detector that can provide arbitrary output during an unbounded time period.

Chandra and Toueg [6] define several classes of failure detectors whose output is a list of *suspected* processes. We say that process p *suspects* process q at a certain point in the execution, if q is included in p's failure detector's output list at this point. To illustrate the upper bound, we focus on a failure detector of class $\Diamond\mathcal{S}$, which has been shown to be the weakest failure detector for solving consensus [5]. A failure detector of this class guarantees two properties:

1. *strong completeness*: there is a time after which every correct process permanently suspects every crashed process; and

[3] Communication steps are sometimes called rounds.

[4] Consensus can be solved even if these assumptions are weakened, as in the *timed asynchronous* model [9], where clock skew is possible but bounded, and likewise, different processing times are possible but their ratio is bounded.

[5] A well-behaved execution is sometimes called *stable*, e.g., in [9]

2. *eventually weak accuracy*: there is a correct process p such that there is a time after which p is not suspected by any correct process.

A *well-behaved* execution in this model is an execution in which there are no failures and no suspicions, that is, all the processes have empty suspicion lists.

A failure detector of class $\diamond\mathcal{S}$ can be implemented in the partial synchrony model defined above using time-outs by increasing the time-out value every time a false suspicion occurs (see [6]). We observe that with this implementation, if the time-out value is initially δ, then in any well-behaved execution of the partial synchrony model, the failure detector generates no suspicions. Therefore, a well-behaved execution of the partial synchrony model can simulate a well-behaved execution of the failure detector model. Thus, we can focus on the failure detector model when showing the upper bound, and on the partial synchrony model when showing the lower bound, and the results would apply to both models.

Another example of a failure detector class is $\diamond\Omega$ [5]. $\diamond\Omega$ is a *leader election* service whose output is the name of one process, presumed to be the leader. Initially, a failure detector of this class can be unreliable: it can name a faulty process as the leader, and can name different leaders at different processes. However, eventually, it must elect a single correct leader, that is, give all the processes the same output, which must be a correct process. In [5], it is shown that $\diamond\Omega$ is equivalent to $\diamond\mathcal{S}$. In a well-behaved execution, a $\diamond\Omega$ failure detector must announce the same correct leader at all the processes from the beginning of the execution.

2.2 Problem Definitions

We now define several variations of the consensus problem.

In all of these problems, every process p_i participating in the algorithm gets an input value v_i, and the algorithm's output is a decision value dec_i. For simplicity, we consider *binary* consensus, that is, we assume that the input values are in $\{0, 1\}$.

An algorithm solves *consensus* if it satisfies the following conditions:

- *Agreement:* If some correct process p_i decides dec_i and another correct process p_j decides dec_j, then $dec_i = dec_j$. That is, no two correct processes decide on different values.
- *Validity:* If a correct process p_i decides dec_i, then there is a process p_j so that $v_j = dec_i$, that is, the decision value is the input value of some process.
- *Termination:* Every correct process eventually decides on some value.

Agreement and validity are *safety* conditions, and termination is a *liveness* condition.

The agreement condition requires correct processes to agree on the same value, but does not require agreement with faulty processes. Uniform consensus strengthens consensus by also requiring agreement with faulty processes, in case they decide before they fail. An algorithm solves *uniform consensus* if it satisfies validity and termination as defined above, along with the following property:

- *Uniform agreement:* If some process p_i decides dec_i and another process p_j decides dec_j, then $dec_i = dec_j$. That is, no two processes decide on different values.

For the sake of the lower bound, we look at a weaker version of consensus, called weak consensus [13]. Weak consensus replaces the validity condition with a weaker one. Namely, an algorithm solves *weak consensus* (resp. *weak uniform consensus*) if it satisfies agreement (resp. uniform agreement) and termination, along with the following property:

- *Weak validity:* For every value $v \in \{0,1\}$ there is a failure-free execution in which some process decides v.

Atomic commit [14] is used in distributed database systems to have multiple database sites agree whether to commit a transaction or to abort it. The decision 1 represents a commit decision, and 0 – abort. Likewise, the input value 1 represents a vote in favor of committing the transaction, the input value 0 represents a vote for aborting it. A commit decision cannot be reached unless all the processes vote in favor of committing the transaction. The *non-blocking atomic commit* problem [28, 3] is a special case of weak uniform consensus. An algorithm solves non-blocking atomic commit if it satisfies uniform agreement and termination, along with the following two properties:

- *Validity:* The decision value can only be 1 if all sites voted 1.
- *Non-Triviality:* If there are no failures and all sites vote 1, then the decision will be 1.

3 The Upper Bound

The literature contains many consensus algorithms for asynchronous models enriched with unreliable failure detectors [6, 26, 16] (or a leader election service [19]). By the FLP [13] result, we know that without enriching the model with such a service, any algorithm satisfying the safety guarantees of consensus is bound to have an infinite undecided execution in which no failures occur. Since such an unreliable failure detector can provide arbitrary output for an unbounded period of time, any algorithm that uses such an unreliable failure detector to circumvent the FLP impossibility result cannot have a finite bound on its running time.

From a practical point of view, however, reality is not so grave. Failure detectors are implemented using time-outs. In practical networks, time-outs can be fine-tuned to be accurate most of the time, and thus, failure detectors typically provide their useful semantics from the very beginning of the execution. Moreover, failures are usually not frequent. Therefore, from a practical perspective, it is interesting to focus on the algorithms' running times in executions in which the underlying service is accurate, and in which no failures occur.

We now illustrate that in such cases, many consensus algorithms terminate within two communication steps. In Section 3.1 we present a simple consensus algorithm that works with $\Diamond\mathcal{S}$, with which in well-behaved executions, process decide within two communication steps. As explained above, $\Diamond\mathcal{S}$ can be implemented in a partial synchrony model in a way that preserves the well-behaved nature of an execution. Therefore, the upper bound also applies to the partial synchrony model.

In Section 3.2 we discuss message omission failures and partitions.

3.1 Consensus with $\Diamond\mathcal{S}$

Several $\Diamond\mathcal{S}$-based algorithms that terminate within two communication steps have been suggested [26, 16, 24]. These algorithms tolerate failures as long as a majority of the processes are correct. They use the rotating coordinator approach [7, 11]. With this approach, the algorithm iterates through a sequence of rounds, at each round, a different coordinator leads the consensus algorithm in order to try and reach a decision. In these algorithms, each process is identified by a unique number from 1 to n, and in round i, the coordinator is process number $(i \bmod n) + 1$. The iterations through multiple rounds is needed in order to account for possible coordinator failures. However, once there is a round in which the coordinator is correct and also not suspected by any process, all the processes decide, and the algorithm terminates.

```
Function Consensus(v_i)
cobegin
 (1) task T1: r_i ← 0; est_i ← v_i; % v_i ≠ ⊥ %
 (2)          while true do
 (3)             c ← (r_i mod n) + 1; est_from_c_i ← ⊥; r_i ← r_i + 1; % round r_i %
 (4)             case (i = c) then est_from_c_i ← est_i
 (5)                  (i ≠ c) then wait until ( (EST(r_i, v) is received from p_c) ∨ (c ∈ suspected_i) );
 (6)                               if (EST(r_i, v) has been received) then est_from_c_i ← v endif
 (7)             endcase;   % est_from_c_i = est_c or ⊥ %
 (8)             ∀j do send EST(r_i, est_from_c_i) to p_j enddo;
 (9)             wait until (EST(r_i, est_from_c) has been received from ⌈(n + 1)/2⌉ processes);
 (10)            let rec_i = {est_from_c such that EST(r_i, est_from_c) has been received at line 5 or 9};
                            % est_from_c = ⊥ or v with v = est_c %
                            % rec_i = {⊥} or {v} or {v, ⊥} %
 (11)            case (rec_i = {⊥})    then skip
 (12)                 (rec_i = {v})    then ∀j ≠ i do send DECIDE(v) to p_j enddo; return(v)
 (13)                 (rec_i = {v, ⊥}) then est_i ← v
 (14)            endcase
 (15)         enddo

 (16) task T2: upon reception of DECIDE(v): ∀j ≠ i do send DECIDE(v) to p_j enddo; return(v)
coend
```

Figure 1: A $\diamond\mathcal{S}$-based Consensus Algorithm

For illustration, we present the algorithm of [24] in Figure 1. In this algorithm, each process p_i keeps track of the round number in a variable r_i, initially zero, and of the estimated consensus value in a variable est_i, initially the process' input value. A round of this algorithm is conducted as follows: the coordinator sends its own estimate to the other processes. Every other process waits until it either receives the estimate from the coordinator, or suspects the current coordinator. Next, all the processes send messages to each other: if a process received the estimate from the coordinator sends this estimate, otherwise, it sends a null value. Finally, each process waits to receive messages from a majority of the processes and then acts as follows:

1. A process that receives only null values proceeds directly to the next round.

2. A process that receives the same non-null value (the coordinator's estimate) from a majority, broadcasts a DECIDE message with this value to all, and decides on this value. That is, the process returns this value.

3. A process that receives some null values and some non-null values sets its estimate est_i to the received non-null value, and proceeds to the next round.

In addition, once a process receives a DECIDE message, it forwards the DECIDE message to all (to account for the case that the coordinator who sent it failed after sending it), and then decides on the value in the DECIDE message.

We informally now argue that the algorithm is correct. For a rigorous proof, see [24]. Validity is ensured because the first coordinator suggests its own initial value, and every other coordinator either suggests its own value or a value that was previously sent by another coordinator. Thus, by induction, only processes' initial values are possible decision values.

To see that agreement is ensured, notice that if some process decides v in round r, then a majority of processes must have sent v in round r. Since every process waits to hear from a majority before proceeding to the next round, every process receives at least one message with v in round r. Thus, every process that proceeds to round $r+1$ sets its own estimate to v beforehand. Clearly, from round $r+1$ onward, v is the only possible decision value.

A decision is made once there is a round in which the coordinator does not fail and is also not suspected by any process. The properties of the $\Diamond\mathcal{S}$ failure detector, together with the fact that the algorithm continuously iterates over all possible coordinators until some process decides, ensure that eventually there is such a round. In addition, once some process decides, every process that decides broadcasts a DECIDE message to all the other processes. This guarantees that if a correct process stops participating in the iterations, all correct processes get its DECIDE message and also decide.

Now, let us consider a well-behaved execution. If the first coordinator does not fail and is also not suspected by any process, then every correct process receives the coordinator's estimate and sends it to the other processes. Moreover, no process sends a null value. Thus, all the processes decide after receiving messages from a majority in the first round, that is, after exactly two communication steps.

3.2 Consensus with Messages Omissions

Consensus is not always solvable in the message omission model. However, if there is a time after which there is a process p so that p's messages to correct processes are not lost, and moreover, there is a majority of correct processes that can communicate reliably with p, then consensus is solvable.

Lamport's Paxos algorithm [19] solves consensus in the asynchronous message omission model, with the above restriction on message loss patterns. Paxos uses a leader election service. In essence, this leader election service is a failure detector of class $\Diamond\Omega$. In executions in which there are no failures, and the leader election service elects a single leader at all the processes from the beginning of the execution, Paxos, as originally presented in [19], requires five communication steps before all processes decide. The first two steps are needed for recovery from past failures, and they do not involve sending actual consensus values. When a sequence of Paxos consensus algorithms are ran, these two steps have to occur only at the beginning or when the leader changes. In a given instance of consensus that is invoked after this phase, only three communication steps are executed. These three steps work in a manner very similar to a round of the $\Diamond\mathcal{S}$-based consensus algorithm presented above.

In the first of these steps, the leader sends its own initial value to all the processes. In the second, all the processes that receive this value record it and send acknowledgments to the leader. Once a majority of the processes has recorded a value, the value is *locked* and no other decision value is possible. Upon receiving acknowledgments from a majority, the leader decides. In the third step, the leader sends the decision value to all the processes and they decide.

A simple variation on Paxos can merge these two steps, by having all the processes send acknowledgments to each other, and every process that receives acknowledgments from a majority decides directly. (This approach was taken in [18]). However, this requires strengthening the restriction on message loss patterns, to require that a majority of processes all communicate reliably with each other, not just with the leader.

4 Synchronous Lower Bounds: Consensus and Uniform Consensus

Consider a synchronous system where at most t processes can crash, and let f be the actual number of failures in an execution. In this section we show a consensus impossibility for the omission model with a single mobile failure. We then show lower bounds for consensus and uniform consensus in the crash failure model.

For the sake of proving the bounds, it will suffice to assume the weak validity property, that is, the bounds will apply to weak consensus and weak uniform consensus.

4.1 Bibliographical Notes

It has been shown that $f+1$ steps are necessary and sufficient for consensus [10], and $f+2$ for uniform consensus, when $f < t-1$. If $f = t-1$ or $f = t$ then both consensus and uniform consensus require $t+1$ steps (see [8]). Dwork and Skeen [12] showed a similar result for atomic commit, which is an instance of weak uniform consensus. Their complexity analysis used a different measure than the one we use here: they modeled the sending of k messages in a single communication step as costing $O(k)$. Clearly, the time bounds obtained this way are higher. However, the two steps bound implicitly appears in this paper.

In the omission model with a single mobile failure no solution at all exists [25]. However, if the adversary is restricted so that eventually there is a majority of the processes that do not experience message omission, consensus is solvable [19]. Moreover, if $t > 1$, then consensus requires two failure-free steps [20].

We describe a technique due to Moses and Rajsbaum [23], and use it to prove all four lower bounds. This technique was shown in [23] to apply to asynchronous message passing and shared memory models, as well as for the synchronous and mobile models we describe here. We show here how to use it also for uniform consensus. Bar-Joseph and Ben-Or [2] use a similar technique for randomized synchronous consensus, and Aguilera and Toueg [1] use it for the synchronous crash failure model.

4.2 Mobile Failures

Proving lower bound and impossibility results can often be thought of as showing that, for any given algorithm, an adversary always has a strategy that can guarantee a desired bad behavior. In the case of the synchronous model, an adversary can choose which processes fail in a round, and what subset of their messages are delivered. We say that such a decision is an *action of the environment.* Assume a (deterministic) consensus algorithm is given. For any initial state of the system the environment actions completely define a execution: the environment defines the messages a process receives in each step, and the protocol of the process defines its new local state and the messages it send next. Formally, an *execution* R is defined in terms of two sequences, r of global states, and α of environment actions, that are interleaved to obtain a computation. Given an execution R (possibly consisting of just one state), let us denote by $R \odot \epsilon$ the execution that results from extending R by having the environment perform the action ϵ. Thus, every execution of can be represented in the Form

$$x \odot \epsilon_1 \odot \epsilon_2 \odot \cdots$$

where x is an initial state for the consensus problem, Con_0, and ϵ_i is an environment action, for every integer $i \geq 0$. An initial state for consensus is determined by the inputs of the processes.

Thus, for each combination of 0's and 1's there is An initial state in Con_0.

To obtain an impossibility result, it is often sufficient to consider only a subset of all possible executions of the model. Thus we define a *system* $\mathcal{S}$ to be a set of executions. A *layering* is a set of environment actions used to generate a system[6]. If the set of layers is chosen appropriately, these executions can have structural properties that will simplify their analysis. For the omission failure model, the following layering works. We use $[k]$ to denote the set $\{1, \ldots, k\}$, with $[0]$ denoting the empty set. The layering consists of nondeterministically choosing a process $i \in \{1, \ldots, n\}$ and a set $[k]$ and performing the action $(i, [k])$, meaning that the messages from i to members of $[k]$ will be lost (and only these will be lost). More precisely, the layering is:

$$\mathsf{L} = \{(i, [k]) : 1 \leq i \leq n \text{ and } 0 \leq k \leq n\}.$$

The impossibility will be obtained by considering the system $\mathcal{S}$ of all executions obtained by repeatedly executing layers from L starting in an initial state. This system has the following critical properties:

1. In $\mathcal{S}$ any process i can crash from any state x on in an execution; simply execute the layer $(i, [n])$ repeatedly after x.

2. At most one process can crash in $\mathcal{S}$, since in any infinite sequence of layers from L at most one process is silenced for an infinite number of steps.

3. Given any state x of an execution of $\mathcal{S}$, there is an execution extending x where no processes crash. For the crash failure model this property is that the only processes that crash are the ones that have already crashed at x (in the mobile failure model, there is no such process).

Since the original asynchronous consensus impossibility [13] it was realized that a crucial role is played by the decision values that are possible in the future of a given global state. This is captured by the notion of the *potential valence* or *potence* for short of a state with respect to a set of executions.

Definition 4.1 *A state x is w-***potent** *with respect to system $\mathcal{S}$ if x is a state of a execution $R \in \mathcal{S}$ in which at least one nonfaulty process decides w. The state x is* **bipotent** *if it is both 0-potent and 1-potent.*

As we shall see, bipotent states play an important role in precluding consensus in the mobile failure model. Our goal will be to show that every consensus algorithm must have an execution whose states are all bipotent. We are interested in bipotent states because an algorithm cannot have terminated in such a state. In the mobile failure model at most one process can crash, but we state the following lemma in a slightly more general way, for t crashes, to use it also in the t-resilient crash model.

Lemma 4.1 *Assume no more than t, $t < n$ processes can fail in an execution. If x is a bipotent state of $\mathcal{S}$ then at least $n - t$ non-failed processes at x have not decided by x.*

Proof: Since x is bipotent, there is an execution R^0 containing x in which at least one nonfaulty process decides 0. The set P_0 of nonfaulty processes in R^0 consists of at least $n - t$ processes,

[6] In asynchronous models a layering consists of sequences of environment actions, and various layerings can be defined to show that consensus is impossible even in executions with very little asynchrony[23]

they are all non-failed at x, and by the agreement property none of them has decided 1 by x. By symmetry of 0 and 1, we obtain the existence of a set P_1 of at least $n-t$ non-failed processes at x that have not decided 0 by x. There is an execution $\hat{R}$ containing x in which the only processes that fail are the ones that are already failed at x (in the mobile failure model, none, since it is always possible to extend a state with a crash-free execution). All processes in $P_0 \cup P_1$ are nonfaulty in $\hat{R}$. By the agreement property, there is at most one value $w \in \{0,1\}$ on which nonfaulty processes decide in $\hat{R}$. If nonfaulty processes do not decide 0 in $\hat{R}$, then no process in P_0 has decided by x, and if they do not decide 1 in $\hat{R}$, then no process in P_1 has decided by x. In either case, the claim holds. ■

The following notion of connectivity can be used to show the existence of bipotent states. With respect to a system $\mathcal{S}$ states x and y are *similar*, denoted by $x \sim_s y$, if there is a process j such that (a) the states x and y are identical except perhaps in the local state of one process j (and it is possible for j to crash), and (b) there exists $i \neq j$ that is non-failed in both x and y. A set of states X is *similarity connected* if the graph $(X, \sim_s)$ induced by $\sim_s$ on X is connected. The following simple lemma is the basis for the impossibility proof.

Lemma 4.2 *Let X be a similarity connected set of states of $\mathcal{S}$ where one crash is possible. If X contains both 0-potent and 1-potent states, then there is a bipotent state in X.*

Proof: Let X^0 be the subset of X consisting of 0-potent states, while X^1 is the subset of 1-potent states. By assumption, both subsets are nonempty. It thus follows that there must be an edge $x^0 \sim_s x^1$ with $x^0 \in X^0$ and $x^1 \in X^1$. The states x^0 and x^1 are identical except perhaps in the local state of one process j (and it is possible for j to crash), and there exists $i \neq j$ that is non-failed in both states. Thus, if j crashes the other processes cannot distinguish between both states, and they all (at least there is one, i) decide the same value. That is, the states x^0 and x^1 have a shared potence. If their shared potence is 0 then x^1 is bipotent, while if the shared potence is 1 then x^0 is bipotent. In either case there is a bipotent state in X, as desired. ■

We can show that the consensus initial states Con_0 are connected in this sense.

Lemma 4.3 Con_0 *is similarity connected with respect to $\mathcal{S}$.*

Proof: Given a state z we denote by z_j the local state of process j in the state z. First, notice that Con_0 is a subset of the states of $\mathcal{S}$. Let $x, y \in \mathsf{Con}_0$, and for every $0 \leq m \leq n$ define x^m by setting $x^m_j = y_j$ for all $j \leq m$, and $x^m_j = x_j$ for all $j > m$. Clearly, $x^m \in \mathsf{Con}_0$, and it is easy to check that $x^0 = x$ and $x^n = y$. Moreover, for every $0 < l \leq n$ we have that x^{m-1} and x^m differ exactly in the local state of process m. Hence, they are similar, and we have that $x^{m-1} \sim_s x^m$. It follows by transitivity that x and y are similarity connected and we are done. ■

We can use Lemma 4.2 and 4.3 to obtain a bipotent initial state x^0.

Lemma 4.4 *The set* Con_0 *contains a bipotent state.*

Proof: Let $x^0 \in \mathsf{Con}_0$ be a 0-potent initial state, and let x^1 be a 1-potent initial state. These exist by the weak validity consensus requirement, and there is an execution starting in x^0 (resp. x^1) where the nonfaulty processes decide 0 (resp. 1). By Lemma 4.3 Con_0 is similarity connected and thus by Lemma 4.2 we obtain that there is a bipotent state in Con_0. ■

It remains to show that the set $\mathsf{L}(x)$ of successors of a state x that can be obtained by applying a layer from L is similarity connected.

Lemma 4.5 *For every state x of $\mathcal{S}$ the set $\mathsf{L}(x)$ is similarity connected.*

Proof: For every pair of processes j, j' we have that $x \cdot (j, [0]) = x \cdot (j', [0])$ because in both cases no messages are lost in the round following x. It follows that $x \cdot (j, [0]) \sim_s x \cdot (j', [0])$. Moreover, for every $k < n$ we have that $x \cdot (j, [k]) \sim_s x \cdot (j, [k+1])$, because the two states differ only in the state of process $k+1$. It follows that $\mathsf{L}(x)$ is similarity connected for all x. ■

A *bipotent* execution is an execution where every state is bipotent.

Lemma 4.6 *There is a bipotent execution in $\mathcal{S}$.*

Proof: By Lemma 4.4, there is a bipotent state, say x^0. We will construct a sequence of bipotent states $x^1, x^2, \ldots, x^k, \ldots$ and a corresponding sequence of layers $L_1, L_2, \ldots, L_k, \ldots$ such that $x^{i+1} = x^i \cdot L_{i+1}$ for all $i \geq 0$. The desired execution will be

$$R_{\mathsf{L}} = x^0 \odot L_1 \odot L_2 \odot \cdots \odot L_k \odot \cdots$$

It remains to define the two sequences. We will define the layers L_i and the states x^i by induction on i. For the basis, x^0 is bipotent. Let $k \geq 0$, and assume that we have constructed sequences $L_1, L_2, \ldots, L_k$ and $x^1, x^2, \ldots, x^k$ with the desired properties. In particular, x^k is a bipotent state. By Lemma 4.5, we have that $\mathsf{L}(x^k)$ is similarity connected. Since x^k is bipotent, $\mathsf{L}(x^k)$ contains both 0-potent and 1-potent states (one process can crash). It follows from Lemma 4.2 that there is a bipotent state $y \in \mathsf{L}(x^k)$. Since $y \in \mathsf{L}(x^k)$ we have by the definition of $\mathsf{L}(x^k)$ that $y = x^k \cdot L$ for some layer $L \in \mathsf{L}$. Set $L_{k+1} = L$ and $x^{k+1} = y$. ■

We conclude the impossibility results as follows. By Lemma 4.6 there is a bipotent execution $R \in \mathsf{L}$. This execution has an infinite number of bipotent states. By Lemma 4.1 no process has decided in a bipotent state (since in this model there are no crashes, and hence there are no failed processes at x). But then no process would ever decide in the execution R contradicting the assumption that the algorithm satisfies the decision property.

Theorem 4.7 *There is no 1-resilient consensus algorithm in the mobile failures model.*

4.3 Synchronous crash failures

The same ideas can be used to show that consensus cannot be reached in a t-resilient synchronous crash failure model before $t+1$ rounds. However, the existence of a bipotent state can be shown to exist only in the first $t-1$ rounds, yielding a t rounds lower bound. The true lower bound of $t+1$ is obtained by showing that two more rounds are still necessary after a bipotent state, by arguing that there is a pair of states x, y, $x \sim_s y$, with different potence. In fact, the same layering works, except that care must be taken to not crash more than t processes in an execution. That is, the action $(j, [k])$ is applicable to a state x only if fewer than t processes are failed at x, and process j is not failed at x. Hence, it will be convenient to have a name for the action where no process fails: clean. This environment action, applicable at all states, involves no new process failure. Messages of failed processes are not delivered, while all messages of non-failed processes are delivered. We denote the layering consisting of all actions of these types by L^t. Notice that the number of processes that fail in an execution of $\mathcal{S}_{\mathsf{L}^t}$ is at most t, since once t processes are failed at a state, all later layers will be clean.

Similar to Lemma 4.5, we have the following. The only difference is that if in x there are already t crashes, then $\mathcal{S}_{\mathsf{L}^t}(x)$ contains just one state, and hence is trivially similarity connected. The proof of part (ii) is using Lemma 4.2, and hence it is needed that no more than $t-2$ processes are failed at x, because then in $\mathsf{L}^t(x)$ at most $t-1$ are failed. Meaning that one process can still fail in states of $\mathsf{L}^t(x)$, and the lemma can be used: to obtain a bipotent state, the possibility of a new crash must exist. Thus we have:

Lemma 4.8 *In the synchronous t-resilient model with crash failures for every state x of $\mathcal{S}_{\mathsf{L}^t}$ the set $\mathsf{L}^t(x)$ is similarity connected. Moreover, if no more than $t-2$ processes are failed at a bipotent state x, then $\mathsf{L}^t(x)$ contains a bipotent state.*

Since Lemma 4.4 guarantees the existence of a bipotent initial state x^0 with 0 failed processes (the proof of that lemma works for any model where at least one process can be silenced), we can use Lemma 4.8 to show the existence of an execution with bipotent states $x^0, x^1, \ldots, x^f$, with one crash per step, for any $f < t$. By Lemma 4.1, this gives us a t-step lower bound for consensus. The true $(t+1)$-round lower bound is obtained by showing that two rounds are still necessary after a bipotent state if $t \leq n-2$:

Lemma 4.9 *Assume that $t \leq n-2$. If $\hat{x}$ is a bipotent state in a layer of $\mathcal{S}_{\mathsf{L}^t}$, then there is a state $y \in \mathsf{L}^t(\hat{x})$ in which at least one non-failed process has not decided.*

Proof: Notice that a state x with t failed processes cannot be bipotent, since there is a unique extension starting at x. Hence, to be bipotent, the state $\hat{x}$ can have no more than $f = t-1$ failed processes. By Lemma 4.8(i), we have that $\mathsf{L}^t(\hat{x})$ is similarity connected. Since $\hat{x}$ is bipotent, there are states $y^0, y^1 \in \mathsf{L}^t(\hat{x})$ such that y^0 is 0-potent and y^1 is 1-potent. The similarity connectivity of $\mathsf{L}^t(\hat{x})$ implies the existence of states $z^0, z^1 \in \mathsf{L}^t(\hat{x})$ (not necessarily distinct) satisfying $z^0 \sim_s z^1$ that are 0- and 1-potent, respectively. Recall that all states of $\mathcal{S}_{\mathsf{L}^t}$ have at most t faulty processes. Since $t \leq n-2$ and z^0 and z^1 agree modulo j for some j, it follows that there is at least one process $i \neq j$ such that i is not failed in both states and $z^0_i = z^1_i$. Assume by way of contradiction that every non failed process is decided in both z^0 and z^1. In particular, i is decided, say with value v. Agreement implies that in both states, every non-failed process decides v. It follows that both z^0 and z^1 are v-potent, and neither of them is $(1-v)$-potent, contradicting the assumption that one of them is 0-potent and the other is 1-potent. ■

We can now put the two results together and obtain the desired lower bound:

Theorem 4.10 *Let $t \leq n-2$. Every t-resilient algorithm for consensus in the synchronous model has an execution in which decision requires at least $t+1$ rounds. Moreover, for every $f \leq t-1$, such an algorithm has an execution with f failures in which decision requires $f+1$ rounds.*

Proof: Case $f \leq t-1$: Since Lemma 4.4 guarantees the existence of a bipotent initial state x^0 with 0 failed processes (the proof of that lemma works for any model where at least one process can be silenced), we can use Lemma 4.8 to show the existence of an execution with bipotent states $x^0, x^1, \ldots, x^f$, with one crash per step, for any f failures, $f \leq t-1$. Lemma 4.1 implies that not all processes have decided in x^f.

Case $f = t$: Now we have the execution with bipotent states $x^0, x^1, \ldots, x^{f-1}$, and we have to use Lemma 4.1, which says that there is a state in the next step, x^f Where not all processes have decided in x^f. ■

Taking $f = 0$ in the previous theorem we get that if one failure is possible and there are at least three processes, one step is necessary to solve consensus:

Corollary 4.11 *Any t-resilient consensus algorithm must perform one communication step before deciding if $1 \geq t \leq n-2$.*

4.4 Uniform Consensus

Now consider a uniform consensus algorithm. We show that one more step is needed in failure-free executions (a special case of the [8] result):

Theorem 4.12 *For $t > 1$, every t-resilient uniform consensus algorithm must perform two communication steps in some failure-free execution before all the processes can decide.*

Proof: Consider an algorithm A solving weak uniform consensus. By the weak validity condition, there is at least one initial state from which the failure-free execution of A decides 0, and another initial state from which the failure-free execution of A decides 1. Therefore, there are two initial states $\hat{x}, \hat{y}$ that differ in exactly the input of one process, j, from which failure-free executions of A decide different values. Assume that in $\hat{x}$ the failure-free execution decides 0, and in $\hat{y}$ – 1.

Assume for the sake of contradiction that all the processes decide after one communication step in all failure-free executions. Consider the state $\hat{x} \cdot (j, [0])$ (where nobody fails). In this state, all processes must decide 0.

Process n has exactly the same state in $\hat{x} \cdot (j, [0])$ and $\hat{x} \cdot (j, [n-1])$. Thus, n decides 0 in $\hat{x} \cdot (j, [n-1])$. By a symmetric argument for $\hat{y}$, n decides 1 in $\hat{y} \cdot (j, [n-1])$. By the uniform agreement requirement, every extension of $\hat{x} \cdot (j, [n-1])$ (and $\hat{y} \cdot (j, [n-1])$) has to decide 0 (decide 1), even if process i crashes. That is, niether $\hat{x} \cdot (j, [n-1])$ nor $\hat{y} \cdot (j, [n-1])$ are bipotent. But n is the only one hearing from j: in $\hat{x} \cdot (j, [n-1]) \cdot (n, [n])$ and $\hat{y} \cdot (j, [n-1]) \cdot (n, [n])$ all correct processes have the same state. So at least one of these states is bipotent, a contradicion. ∎

5 Deriving Lower Bounds for Non-Synchronous Models

We now use the results of the previous section to derive the lower bounds for consensus in the of partial synchrony and asynchronous with unreliable failure detectors. We do this in two steps: first, in Section 5.1 we derive a lower bound for uniform consensus in these models; next, in Section 5.2 we show that any algorithm that solves uniform consensus in these models also solves consensus. The lower bound for consensus follows.

5.1 A lower bound for uniform consensus

In the previous section, we considered a synchronous model where up to t processes may fail. In that model, we have shown that any uniform consensus algorithm for the synchronous model where at least two processes can fail must perform two communication steps in some failure-free executions before all the processes can decide. This lower bound also applies to well-behaved executions in the partial synchrony model and in the asynchronous model with failure detectors as we show in the two lemmas below.

Lemma 5.1 *For every algorithm that solves uniform consensus in the partial synchrony model where at least two processes can crash, there is a well-behaved execution in which the algorithm performs at least two communication steps before all the processes decide.*

Proof: Assume the proposition is false, then there exists a uniform consensus algorithm that tolerates two crashes and decides after at most one communication step in every well-behaved execution, that is, in every failure-free synchronous execution. Clearly, this algorithm solves uniform consensus in the synchronous model, and decides within one communication step in every failure-free execution. A contradiction. ∎

Lemma 5.2 *For every algorithm that solves uniform consensus in the asynchronous model enriched with failure detector $\Diamond\mathcal{S}$ or $\Diamond\Omega$, where at least two processes can crash, there is a well-behaved execution in which the algorithm performs at least two communication steps before all the processes decide.*

Proof: It suffices to show that we can implement $\Diamond\mathcal{S}$ and $\Diamond\Omega$ failure detectors in the synchronous model, so that their execution is well-behaved in failure-free executions of the synchronous model.

We construct a $\Diamond\mathcal{S}$ failure detector using the following algorithm: The algorithm is conducted in synchronous communication steps. In every step, every process sends a message to every other process, and waits δ time until the next step. Initially, every process p has an empty suspect list. At every subsequent communication step, p suspects processes from which it did not get messages in this step. Clearly, in the synchronous model, this failure detector suspects a process if and only if the process has crashed; this is the *perfect* failure detector defined in [6]. In particular, it is also a failure detector of class $\Diamond\mathcal{S}$. Moreover, in failure-free executions, this failure detector does not suspect any process. Thus, a failure-free execution of the synchronous model simulates a well-behaved execution of the asynchronous model with a $\Diamond\mathcal{S}$ failure detector.

To implement a $\Diamond\Omega$ failure detector, we modify the above algorithm to have each process output the process with the lowest identifier among the processes it does not suspect. Since the failure detector suspects a process if and only if the process has crashed, the failure detector at each process outputs the first non-crashed process (in lexicographical order). Thus, after the last failure occurs, all the processes elect the same correct leader. In failure-free executions, the leader is always process 1, from the beginning of the execution. Thus, this is a well-behaved execution of $\Diamond\Omega$. ∎

5.2 From consensus to uniform consensus

Guerraoui [15] proves that any algorithm that solves consensus in an asynchronous model enriched with certain classes of failure detectors, including $\Diamond\mathcal{S}$, also solves uniform consensus. Below, we paraphrase Guerraoui's proof, applying it to the models we consider in this paper, namely, the partial synchrony model, and the asynchronous model enriched with any failure detector that can give arbitrary output for an arbitrary prefix of the execution.

Lemma 5.3 *In a partial synchrony model or in an asynchronous model with a failure detector that can give arbitrary output for an arbitrary prefix of the execution, any algorithm that solves consensus also solves uniform consensus.*

Proof: Assume that there is an algorithm that solves consensus but not uniform consensus in one of the aforementioned models. Then, the algorithm has an execution e in which two processes p_i and p_j decide on different values, and at least one of them fails after deciding. Assume, without loss of generality, that p_i decides before p_j. Let t_i be the point in the execution e at which p_i decides, and t_j, the point at which p_j decides. Note that until point t_i, neither p_i nor p_j fails, and until t_j, p_j does not fail.

We now construct an execution e' in which both p_i and p_j are correct, as follows: until point t_i, e is identical to e'. Between t_i and t_j, p_i may send in e' messages that it does not send in e

(because it may be crashed during part of this time in e and not in e'). We delay the receipt of such messages by any process until after point t_j in the execution[7]. In addition, we have the failure detector oracles at all the processes give the same outputs in the prefix of e' until t_j as in in the prefix of e until t_j. After time t_j, we have the failure detector satisfy the required semantics. Since the failure detector can give arbitrary output for an arbitrary prefix of the execution, we can delay this prefix until time t_j.

Observe that e' is indistinguishable from e for process p_i in the prefix until t_i, because e and e' are identical until that point. Therefore, p_i decides in e' at time t_i the same value that it does in e. Observe also that for every process other than p_i, e' is indistinguishable from e until time t_j. Therefore, p_j decides in e' at time t_j the same value that it does in e. Thus p_i and p_j decide upon different values in e', although both are correct. A contradiction. ■

We conclude with the following theorem:

Theorem 5.4 *Consider a model of partial synchrony or an asynchronous model enriched with a $\Diamond\mathcal{S}$ or $\Diamond\Omega$ failure detector, where at least two processes can crash. For every algorithm that solves consensus in these models, there is a well-behaved execution in which the algorithm performs at least two communication steps before all the processes decide.*

6 Conclusions, on an Optimistic Note

We have seen that in practical models where messages may be occasionally lost or delayed, fault-tolerant consensus algorithms require two communication steps even in well-behaved executions with synchronous communication and no failures or false failure suspicions. This is in contrast to algorithms for consensus in the synchronous crash failure model, which can decide in one communication step in failure-free executions. The additional communication step is required in such practical models due to the fact that, in these models, correct processes may be mistaken for faulty ones. We observe that the additional communication step is needed in order to avoid disagreement with processes that are incorrectly suspected to have failed. This observation can give insight to the possible gains of using an *optimistic* approach in lieu of consensus, as we explain below.

The agreement condition of consensus is conservative: it requires that two correct processes never decide on distinct values. Thus, when used for state machine replication [21, 27], consensus never introduces inconsistencies among correct replica. In other words, consensus guarantees *serializability*, that is, in every execution, all the replica exhibit the same sequence of states. In contrast, an optimistic approach allows replica to temporarily diverge, as long as conflicts can later be detected and reconciled. Optimistic approaches can be used if the application can live with temporary conflicts,. These approaches are generally cost-effective if conflicts are rare.

Group communication systems, e.g., Isis [4] and Amoeba [17], implement optimistic algorithms for consensus, or state machine replication. These algorithms operate in a single communication step, using leader election. In these algorithms, the leader sends its own proposed value to all the processes, and decides upon this value. All the recipients of the leader's message immediately decide upon the suggested value. As long as the leader is not suspected by any of the processes, consistency is preserved. When the leader is suspected, a new leader is elected, and the new leader attempts to learn the decision value from the processes it can communicate with. As long as there are no false suspicions, consistency is ensured. However, if the leader is correct but suspected to

[7]Recall that in asynchronous or partial synchrony models, the environment can delay messages arbitrarily long for an arbitrary prefix of the execution, that is, the global stabilization time (GST) can occur after t_j.

have crashed, the other processes can decide upon a different value than the leader. Isis resolves such inconsistencies by forcing the suspected leader to fail and re-incarnate itself as a new process, whereby it adopts the state of the state of the other replica.

In general, conflicts are inevitable when a single-communication step algorithm is used in practical models. This is due to the lower bound proven in this paper. On the other hand, conflicts can be detected immediately when processes that at some point suspected each other re-establish the communication between them. Thus, conflicts can be reconciled quickly. The cost-effectiveness of optimism depends on the rate of conflicts. Recall the observation that the role of the extra communication step in practical models is to avoid potential conflicts with processes that are incorrectly suspected to have failed. This suggests that the frequency of conflicts occurring with the optimistic approach depends on the frequency of false suspicions. Thus, the rate of false suspicions in a given system can be a guideline for deciding when to use optimism, and when a conservative approach (like consensus) would be preferable.

References

[1] M. K. Aguilera and S. Toueg. A simple bivalency-based proof that t-resilient consensus requires $t+1$ rounds. *Inf. Process. Lett.*, 71(3-4):155–158, 1999.

[2] Z. Bar-Joseph and M. Ben-Or. A tight lower bound for randomized synchronous consensus. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 193–199, 1998.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[5] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 147–158, 1992.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[7] J. Chang and N. Maxemchunk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3), 1984.

[8] B. Charron-Bost and A. Schiper. Uniform consensus is harder than consensus (extended abstract). Technical Report DSC/2000/028, Swiss Federal Institute of Technology, Lausanne, Switzerland, May 2000.

[9] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, June 1999.

[10] D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in byzantine agreement. *J. ACM*, 37(4):720–741, October 1990.

[11] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[12] C. Dwork and D. Skeen. The inherent cost of nonblocking atomic commitment. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–11, 1983.

[13] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.

[14] J. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, volume 60, pages 393–481. Springer Verlag, Berlin, 1978.

[15] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In J.-M. Hélary and M. Raynal, editors, *9th International Workshop on Distributed Algorithms (WDAG)*, pages 87–100. Springer Verlag, September 1995. LNCS 972.

[16] M. Hufrin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4), 1999.

[17] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *11th International Conference on Distributed Computing Systems (ICDCS)*, pages 882–891, May 1991.

[18] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[19] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. Also Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

[20] L. Lamport. Lower bounds on consensus. Unpublished manuscript, March 2000.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 78.

[22] B. Lampson. How to build a highly available system using consensus. In Babaoglu and Marzullo, editors, *Distributed Algorithms*, LNCS 1151. Springer-Verlag, 1996.

[23] Y. Moses and S. Rajsbaum. The unified structure of consensus: a layered analysis approach. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 123–132. ACM, June 1998. submitted for journal publication.

[24] A. Mostefaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: a general approach. In *13th International Symposium on DIStributed Computing (DISC)*, Bratislava, Slovak Republic, 1999.

[25] N. Santoro and P. Widmayer. Time is not a healer. In *6th Annual Symp. Theor. Aspects of Computer Science*, volume 349 of *LNCS*, pages 304–313, Paderborn, Germany, Feb. 1989. Springer Verlag.

[26] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[27] F. B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[28] D. Skeen. Nonblocking commit protocols. In *ACM SIGMOD International Symposium on Management of Data*, pages 133–142, 1981.